NAME

gitprotocol-http - Git HTTP-based protocols

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

Git supports two HTTP based transfer protocols. A "dumb" protocol which requires only a standard HTTP server on the server end of the connection, and a "smart" protocol which requires a Git aware CGI (or server module). This document describes both protocols.

As a design feature smart clients can automatically upgrade "dumb" protocol URLs to smart URLs. This permits all users to have the same published URL, and the peers automatically select the most efficient transport available to them.

URL FORMAT

URLs for Git repositories accessed by HTTP use the standard HTTP URL syntax documented by RFC 1738, so they are of the form:

http://<host>:<port>/<path>?<searchpart>

Within this documentation the placeholder **$GIT_URL** will stand for the http:// repository URL entered by the end-user.

Servers SHOULD handle all requests to locations matching **$GIT_URL**, as both the "smart" and "dumb" HTTP protocols used by Git operate by appending additional path components onto the end of the user supplied **$GIT_URL** string.

An example of a dumb client requesting for a loose object:

$GIT_URL: http://example.com:8080/git/repo.git
URL request: http://example.com:8080/git/repo.git/objects/d0/49f6c27a2244e12041955e262a404c7faba355

An example of a smart request to a catch-all gateway:

$GIT_URL: http://example.com/daemon.cgi?svc=git&q=
URL request: http://example.com/daemon.cgi?svc=git&q=/info/refs&service=git-receive-pack

An example of a request to a submodule:

$GIT_URL: http://example.com/git/repo.git/path/submodule.git
URL request: http://example.com/git/repo.git/path/submodule.git/info/refs

Clients MUST strip a trailing /, if present, from the user supplied **$GIT_URL** string to prevent empty path tokens (//) from appearing in any URL sent to a server. Compatible clients MUST expand **$GIT_URL/info/refs** as **foo/info/refs** and not **foo//info/refs**.

AUTHENTICATION

Standard HTTP authentication is used if authentication is required to access a repository, and MAY be configured and enforced by the HTTP server software.

Because Git repositories are accessed by standard path components server administrators MAY use directory based permissions within their HTTP server to control repository access.

Clients SHOULD support Basic authentication as described by RFC 2617. Servers SHOULD support Basic authentication by relying upon the HTTP server placed in front of the Git server software.

Servers SHOULD NOT require HTTP cookies for the purposes of authentication or access control.

Clients and servers MAY support other common forms of HTTP based authentication, such as Digest authentication.

SSL

Clients and servers SHOULD support SSL, particularly to protect passwords when relying on Basic HTTP authentication.

SESSION STATE

The Git over HTTP protocol (much like HTTP itself) is stateless from the perspective of the HTTP server side. All state MUST be retained and managed by the client process. This permits simple round-robin load-balancing on the server side, without needing to worry about state management.

Clients MUST NOT require state management on the server side in order to function correctly.

Servers MUST NOT require HTTP cookies in order to function correctly. Clients MAY store and forward HTTP cookies during request processing as described by RFC 2616 (HTTP/1.1). Servers SHOULD ignore any cookies sent by a client.

GENERAL REQUEST PROCESSING

Except where noted, all standard HTTP behavior SHOULD be assumed by both client and server. This includes (but is not necessarily limited to):

If there is no repository at **$GIT_URL**, or the resource pointed to by a location matching **$GIT_URL** does not exist, the server MUST NOT respond with **200 OK** response. A server SHOULD respond with **404 Not Found**, **410 Gone**, or any other suitable HTTP status code which does not imply the resource exists as requested.

If there is a repository at **$GIT_URL**, but access is not currently permitted, the server MUST respond with the **403 Forbidden** HTTP status code.

Servers SHOULD support both HTTP 1.0 and HTTP 1.1. Servers SHOULD support chunked encoding for both request and response bodies.

Clients SHOULD support both HTTP 1.0 and HTTP 1.1. Clients SHOULD support chunked encoding for both request and response bodies.

Servers MAY return ETag and/or Last-Modified headers.

Clients MAY revalidate cached entities by including If-Modified-Since and/or If-None-Match request headers.

Servers MAY return **304 Not Modified** if the relevant headers appear in the request and the entity has not changed. Clients MUST treat **304 Not Modified** identical to **200 OK** by reusing the cached entity.

Clients MAY reuse a cached entity without revalidation if the Cache-Control and/or Expires header permits caching. Clients and servers MUST follow RFC 2616 for cache controls.

DISCOVERING REFERENCES

All HTTP clients MUST begin either a fetch or a push exchange by discovering the references available on the remote repository.

Dumb Clients

HTTP clients that only support the "dumb" protocol MUST discover references by making a request for the special info/refs file of the repository.

Dumb HTTP clients MUST make a **GET** request to **$GIT_URL/info/refs**, without any search/query parameters.

```
C: GET $GIT_URL/info/refs HTTP/1.0

S: 200 OK
S:
```

```
S: 95dcfa3633004da0049d3d0fa03f80589cbcaf31  refs/heads/maint
S: d049f6c27a2244e12041955e262a404c7faba355  refs/heads/master
S: 2cb58b79488a98d2721cea644875a8dd0026b115  refs/tags/v1.0
S: a3c2e2402b99163d1d59756e5f207ae21cccba4c  refs/tags/v1.0^{}
```

The Content-Type of the returned info/refs entity SHOULD be **text/plain; charset=utf-8**, but MAY be any content type. Clients MUST NOT attempt to validate the returned Content-Type. Dumb servers MUST NOT return a return type starting with **application/x-git-**.

Cache-Control headers MAY be returned to disable caching of the returned entity.

When examining the response clients SHOULD only examine the HTTP status code. Valid responses are **200 OK**, or **304 Not Modified**.

The returned content is a UNIX formatted text file describing each ref and its known value. The file SHOULD be sorted by name according to the C locale ordering. The file SHOULD NOT include the default ref named **HEAD**.

```
info_refs   =  *( ref_record )
ref_record  =  any_ref / peeled_ref

any_ref     =  obj-id HTAB refname LF
peeled_ref  =  obj-id HTAB refname LF
               obj-id HTAB refname "^{}" LF
```

Smart Clients

HTTP clients that support the "smart" protocol (or both the "smart" and "dumb" protocols) MUST discover references by making a parameterized request for the info/refs file of the repository.

The request MUST contain exactly one query parameter, **service=$servicename**, where **$servicename** MUST be the service name the client wishes to contact to complete the operation. The request MUST NOT contain additional query parameters.

```
C: GET $GIT_URL/info/refs?service=git-upload-pack HTTP/1.0
```

dumb server reply:

```
S: 200 OK
S:
S: 95dcfa3633004da0049d3d0fa03f80589cbcaf31  refs/heads/maint
```

S: d049f6c27a2244e12041955e262a404c7faba355 refs/heads/master
S: 2cb58b79488a98d2721cea644875a8dd0026b115 refs/tags/v1.0
S: a3c2e2402b99163d1d59756e5f207ae21cccba4c refs/tags/v1.0^{}

smart server reply:

S: 200 OK
S: Content-Type: application/x-git-upload-pack-advertisement
S: Cache-Control: no-cache
S:
S: 001e# service=git-upload-pack\n
S: 0000
S: 004895dcfa3633004da0049d3d0fa03f80589cbcaf31 refs/heads/maint\0multi_ack\n
S: 003fd049f6c27a2244e12041955e262a404c7faba355 refs/heads/master\n
S: 003c2cb58b79488a98d2721cea644875a8dd0026b115 refs/tags/v1.0\n
S: 003fa3c2e2402b99163d1d59756e5f207ae21cccba4c refs/tags/v1.0^{}\n
S: 0000

The client may send Extra Parameters (see **gitprotocol-pack**(5)) as a colon-separated string in the Git-Protocol HTTP header.

Uses the **--http-backend-info-refs** option to **git-upload-pack**(1).

Dumb Server Response

Dumb servers MUST respond with the dumb server reply format.

See the prior section under dumb clients for a more detailed description of the dumb server response.

Smart Server Response

If the server does not recognize the requested service name, or the requested service name has been disabled by the server administrator, the server MUST respond with the **403 Forbidden** HTTP status code.

Otherwise, smart servers MUST respond with the smart server reply format for the requested service name.

Cache-Control headers SHOULD be used to disable caching of the returned entity.

The Content-Type MUST be **application/x-$servicename-advertisement**. Clients SHOULD fall back to the dumb protocol if another content type is returned. When falling back to the dumb protocol clients SHOULD NOT make an additional request to **$GIT_URL/info/refs**, but instead SHOULD use the response already in hand. Clients MUST NOT continue if they do not support the dumb protocol.

Clients MUST validate the status code is either **200 OK** or **304 Not Modified**.

Clients MUST validate the first five bytes of the response entity matches the regex **^[0-9a-f]{4}#**. If this test fails, clients MUST NOT continue.

Clients MUST parse the entire response as a sequence of pkt-line records.

Clients MUST verify the first pkt-line is **# service=$servicename**. Servers MUST set $servicename to be the request parameter value. Servers SHOULD include an LF at the end of this line. Clients MUST ignore an LF at the end of the line.

Servers MUST terminate the response with the magic **0000** end pkt-line marker.

The returned response is a pkt-line stream describing each ref and its known value. The stream SHOULD be sorted by name according to the C locale ordering. The stream SHOULD include the default ref named **HEAD** as the first ref. The stream MUST include capability declarations behind a NUL on the first ref.

The returned response contains "version 1" if "version=1" was sent as an Extra Parameter.

```
smart_reply     =  PKT-LINE("# service=$servicename" LF)
                   "0000"
                   *1("version 1")
                   ref_list
                   "0000"
ref_list        =  empty_list / non_empty_list

empty_list      =  PKT-LINE(zero-id SP "capabilities^{}" NUL cap-list LF)

non_empty_list  =  PKT-LINE(obj-id SP name NUL cap_list LF)
                   *ref_record

cap-list        =  capability *(SP capability)
capability      =  1*(LC_ALPHA / DIGIT / "-" / "_")
```

```
LC_ALPHA       =  %x61-7A

ref_record     =  any_ref / peeled_ref
any_ref        =  PKT-LINE(obj-id SP name LF)
peeled_ref     =  PKT-LINE(obj-id SP name LF)
                  PKT-LINE(obj-id SP name "^{}" LF
```

SMART SERVICE GIT-UPLOAD-PACK

This service reads from the repository pointed to by **$GIT_URL**.

Clients MUST first perform ref discovery with **$GIT_URL/info/refs?service=git-upload-pack**.

```
C: POST $GIT_URL/git-upload-pack HTTP/1.0
C: Content-Type: application/x-git-upload-pack-request
C:
C: 0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7\n
C: 0032have 441b40d833fdfa93eb2908e52742248faf0ee993\n
C: 0000

S: 200 OK
S: Content-Type: application/x-git-upload-pack-result
S: Cache-Control: no-cache
S:
S: ....ACK %s, continue
S: ....NAK
```

Clients MUST NOT reuse or revalidate a cached response. Servers MUST include sufficient Cache-Control headers to prevent caching of the response.

Servers SHOULD support all capabilities defined here.

Clients MUST send at least one "want" command in the request body. Clients MUST NOT reference an id in a "want" command which did not appear in the response obtained through ref discovery unless the server advertises capability **allow-tip-sha1-in-want** or **allow-reachable-sha1-in-want**.

```
compute_request   =  want_list
                     have_list
                     request_end
request_end       =  "0000" / "done"
```

```
want_list       =  PKT-LINE(want SP cap_list LF)
                  *(want_pkt)
want_pkt        =  PKT-LINE(want LF)
want            =  "want" SP id
cap_list        =  capability *(SP capability)

have_list       =  *PKT-LINE("have" SP id LF)
```

TODO: Document this further.

The Negotiation Algorithm

The computation to select the minimal pack proceeds as follows (C = client, S = server):

init step:

C: Use ref discovery to obtain the advertised refs.

C: Place any object seen into set **advertised**.

C: Build an empty set, **common**, to hold the objects that are later determined to be on both ends.

C: Build a set, **want**, of the objects from **advertised** the client wants to fetch, based on what it saw during ref discovery.

C: Start a queue, **c_pending**, ordered by commit time (popping newest first). Add all client refs. When a commit is popped from the queue its parents SHOULD be automatically inserted back. Commits MUST only enter the queue once.

one compute step:

C: Send one **$GIT_URL/git-upload-pack** request:

```
C: 0032want <want #1>.....................
C: 0032want <want #2>.....................
....
C: 0032have <common #1>.....................
C: 0032have <common #2>.....................
....
C: 0032have <have #1>.....................
C: 0032have <have #2>.....................
```

....
C: 0000

The stream is organized into "commands", with each command appearing by itself in a pkt-line. Within a command line, the text leading up to the first space is the command name, and the remainder of the line to the first LF is the value. Command lines are terminated with an LF as the last byte of the pkt-line value.

Commands MUST appear in the following order, if they appear at all in the request stream:

The stream is terminated by a pkt-line flush (**0000**).

A single "want" or "have" command MUST have one hex formatted object name as its value. Multiple object names MUST be sent by sending multiple commands. Object names MUST be given using the object format negotiated through the **object-format** capability (default SHA-1).

The **have** list is created by popping the first 32 commits from **c_pending**. Less can be supplied if **c_pending** empties.

If the client has sent 256 "have" commits and has not yet received one of those back from **s_common**, or the client has emptied **c_pending** it SHOULD include a "done" command to let the server know it won’t proceed:

C: 0009done

S: Parse the git-upload-pack request:

Verify all objects in **want** are directly reachable from refs.

The server MAY walk backwards through history or through the reflog to permit slightly stale requests.

If no "want" objects are received, send an error: TODO: Define error if no "want" lines are requested.

If any "want" object is not reachable, send an error: TODO: Define error if an invalid "want" is requested.

Create an empty list, **s_common**.

If "have" was sent:

Loop through the objects in the order supplied by the client.

For each object, if the server has the object reachable from a ref, add it to **s_common**. If a commit is added to **s_common**, do not add any ancestors, even if they also appear in **have**.

S: Send the git-upload-pack response:

If the server has found a closed set of objects to pack or the request ends with "done", it replies with the pack. TODO: Document the pack based response

S: PACK...

The returned stream is the side-band-64k protocol supported by the git-upload-pack service, and the pack is embedded into stream 1. Progress messages from the server side MAY appear in stream 2.

Here a "closed set of objects" is defined to have at least one path from every "want" to at least one "common" object.

If the server needs more information, it replies with a status continue response: TODO: Document the non-pack response

C: Parse the upload-pack response: TODO: Document parsing response

Do another compute step.

SMART SERVICE GIT-RECEIVE-PACK

This service reads from the repository pointed to by **$GIT_URL**.

Clients MUST first perform ref discovery with **$GIT_URL/info/refs?service=git-receive-pack**.

C: POST $GIT_URL/git-receive-pack HTTP/1.0
C: Content-Type: application/x-git-receive-pack-request
C:
C: ....0a53e9ddeaddad63ad106860237bbf53411d11a7 441b40d833fdfa93eb2908e52742248faf0ee993 refs/heads/m
C: 0000

```
C: PACK....

S: 200 OK
S: Content-Type: application/x-git-receive-pack-result
S: Cache-Control: no-cache
S:
S: ....
```

Clients MUST NOT reuse or revalidate a cached response. Servers MUST include sufficient Cache-Control headers to prevent caching of the response.

Servers SHOULD support all capabilities defined here.

Clients MUST send at least one command in the request body. Within the command portion of the request body clients SHOULD send the id obtained through ref discovery as old_id.

```
update_request  =  command_list
                   "PACK" <binary data>

command_list    =  PKT-LINE(command NUL cap_list LF)
                   *(command_pkt)
command_pkt     =  PKT-LINE(command LF)
cap_list        =  *(SP capability) SP

command         =  create / delete / update
create          =  zero-id SP new_id SP name
delete          =  old_id SP zero-id SP name
update          =  old_id SP new_id SP name
```

TODO: Document this further.

REFERENCES

RFC 1738: Uniform Resource Locators (URL)[1] **RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1**[2]

SEE ALSO

gitprotocol-pack(5) **gitprotocol-capabilities**(5)

GIT

Part of the **git**(1) suite

NOTES

1. RFC 1738: Uniform Resource Locators (URL)
   http://www.ietf.org/rfc/rfc1738.txt

2. RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1
   http://www.ietf.org/rfc/rfc2616.txt